Package: blockr.process (via r-universe)

November 19, 2026

Title Process Orchestration Blocks for blockr

Version 0.2.1.9001

URL https://cynkra.github.io/blockr.process/, https://github.com/cynkra/blockr.process

BugReports https://github.com/cynkra/blockr.process/issues

Description The organization as a data frame: process definitions, instances, and tasks live in one wide table that flows through a blockr board, in BPMN vocabulary throughout (process, task, lane, multi-instance, collection, element, assignee). A process block emits the table, task blocks write an append-only event log, a headless worker runs the tasks that are scripts, a file inbox lets other systems move a process forward, and a BPMN block renders the table as a standards-conform process diagram with live status painted on. The package is two layers in one namespace: the engine (table semantics, event store, worker, inbox) is pure R with no UI dependency, the blocks sit on top, and a test enforces the direction -- see 'dev/architecture.md'. The BPMN half (model, interchange XML, auto-layout, widget) lives in the 'bpmn-*.R' files and depends on nothing else here, so either layer can be lifted out into a package of its own once the process model stops moving.

License GPL (>= 3)

Encoding UTF-8

Imports blockr.core, blockr.outline (>= 0.0.107), blockr.dplyr (>= 0.2.0), htmltools, htmlwidgets, jsonlite, shiny, stats, tibble, utils, xml2

Suggests blockr.dock, callr, knitr, rmarkdown, shinytest2, testthat (>= 3.0.0), withr

Remotes BristolMyersSquibb/blockr.dplyr, cynkra/blockr.outline

VignetteBuilder knitr

Config/testthat/edition 3

RoxygenNote 7.3.3

Config/pak/sysreqs cmake make libicu-dev libuv1-dev libxml2-dev zlib1g-dev

Repository https://cynkra.r-universe.dev

Date/Publication 2026-11-15 18:42:35 UTC

RemoteUrl https://github.com/cynkra/blockr.process

RemoteRef HEAD

RemoteSha 89092ba973c6354b36e813ba8deb90d69c0cc790

Contents

apply_events . . . 3
as_bpmn . . . 3
bpmn . . . 4
bpmn_to_table . . . 6
bpmn_widget . . . 7
bpmn_widget-shiny . . . 7
bpmn_xml . . . 8
code_version . . . 8
example_process . . . 9
expand_instance . . . 9
inbox_dir . . . 10
ingest_inbox . . . 10
instance_collection . . . 11
instance_definition . . . 12
instance_event . . . 12
instance_events . . . 13
instance_latest . . . 14
instance_log . . . 14
instance_status . . . 15
instance_table . . . 15
instance_view . . . 16
layout_bpmn . . . 16
new_assign_block . . . 17
new_bpmn_block . . . 17
new_event_log_block . . . 18
new_process_block . . . 18
new_runstate_block . . . 19
new_start_instance_block . . . 20
new_tasks_block . . . 21
process_act . . . 22
process_groups . . . 23
process_interpreters . . . 24
process_status . . . 25
process_tasks . . . 25
process_version . . . 26
read_bpmn . . . 26

run_worker 27
start_instance 29
worker_alive 29
write_bpmn 30
write_inbox_message 31

Index **32**

`apply_events` *Overlay an instance's events onto a process definition*

Description

The counterpart of [apply_assignments()]: same result shape, but the instance columns come from the store instead of from block state. Exported because the instance state block's expression calls it at evaluation time.

Usage

```
apply_events(df, store = ".runs", instance = "instance", stamp = NULL)
```

Arguments

`df`	A wide process table (the definition).
`store`	Store directory.
`instance`	Instance id to filter by, or 'NULL' for all instances.
`stamp`	Poll counter (ignored).

Value

The wide table with 'assignee', 'due', 'status' and 'instance' applied.

`as_bpmn` *Convert a wide process table to a tidy BPMN model*

Description

The wide table is the "one table" convention: one row per process task, skeleton columns describing the task, optional instance columns ('assignee', 'due', 'status') describing a run. 'as_bpmn()' reads only the skeleton columns.

Usage

```
as_bpmn(df, name = "Process", external = NULL)
```

Arguments

`df`	Data frame with at least ‘task‘ and ‘name‘ columns; ‘role‘, ‘depends_on‘, ‘script‘, ‘parent‘, ‘collection‘, ‘join‘, ‘complete_when‘ and ‘sequential‘ are used when present.
`name`	Process name.
`external`	Optional label of the external pool the inbox messages come from (e.g. ‘"Reporting unit"‘). When set, every receive task (‘role == "system"‘ without a script) gets a message flow from a collapsed pool of that name, so the diagram shows where the process touches the outside world. ‘NULL‘ (the default) draws no pool.

Details

Conventions: - ‘role == "system"‘ becomes a script task when ‘script‘ is set, and a **receive task** when it is not: a system task without a script is completed by an inbox message (see ‘vignette("external-systems")‘), and the envelope marker says so. Any other non-empty role becomes a user task; empty role a plain task. - A row with a non-empty ‘collection‘ is a **multi-instance sub-process** (see [process_groups()]), not work of its own: it is dropped from the diagram and the rows whose ‘parent‘ names it are marked multi-instance instead, which is the ‘|||‘ marker on the activity. - ‘depends_on‘ is comma-separated. A dependency may qualify the outcome it waits for: ‘"qacheck:true"‘. The qualifier becomes the label on the arrow (‘true‘/‘false‘ is the convention for binary checks). - A dependency may name a **container**, in either direction: ‘"each_g"‘ waits for the sub-process to complete, and a container's own ‘depends_on‘ gates its members. Sequence flow may not cross a sub-process boundary in BPMN, so [process_body()] lowers container edges onto the members before anything else reads them (see there). - Start and end events are synthesized: tasks without dependencies hang off a start event, tasks nothing depends on flow into an end event. - Fan-in of more than one (non-loop) dependency synthesizes a join gateway. ‘join‘ on the waiting row says which kind: ‘all‘ (the default, a parallel gateway), ‘any‘, or a quorum like ‘n=3‘ / ‘pct=90‘ (see [process_quorum()]). Loop-back edges (a dependency that is itself reachable from the task) stay direct arrows, so rework loops render as loops.

Value

A [bpmn()] model.

`bpmn` *Create a tidy BPMN model*

Description

A BPMN process model as two data frames: one row per node (events, tasks, gateways), one row per sequence flow (the arrows). This is the tidy counterpart of the semantic half of a BPMN 2.0 XML file. Layout (the "diagram interchange" half) is computed automatically and is never part of the model.

Usage

```
bpmn(nodes, flows, name = "Process", messages = NULL)
```

Arguments

nodes	Data frame with columns ‘id‘, ‘name‘, ‘type‘ and optionally ‘lane‘, ‘script‘ (what a script task runs, kept for round-tripping through [bpmn_to_table()]), ‘multi‘ (logical: draw the activity as multi-instance, the ‘|||‘ marker) and ‘multi_seq‘ (logical: that multi-instance runs its elements one at a time, BPMN ‘isSequential‘, drawn as the ‘ ‘ marker). Recognized types (aliases in parentheses): ‘startEvent‘ (‘start‘), ‘endEvent‘ (‘end‘), ‘task‘, ‘userTask‘ (‘user‘), ‘serviceTask‘ (‘service‘), ‘scriptTask‘ (‘script‘), ‘manualTask‘ (‘manual‘), ‘sendTask‘ (‘send‘), ‘receiveTask‘ (‘receive‘, a task an incoming message completes), ‘exclusiveGateway‘ (‘gateway‘, ‘xor‘), ‘parallelGateway‘ (‘parallel‘, ‘and‘), ‘inclusiveGateway‘ (‘inclusive‘, ‘or‘), ‘complexGateway‘ (‘complex‘, the quorum join).
flows	Data frame with columns ‘from‘, ‘to‘ and optionally ‘name‘ (the label on the arrow, e.g. a gateway condition).
name	Process name.
messages	Optional data frame with columns ‘from‘ and ‘to‘: message flows between the process and the outside world. Exactly one side of each row must be a node id; the other side is the label of an external pool (drawn collapsed), e.g. ‘data.frame(from = ”Reporting unit”, to = ”deliver”)‘ for an incoming delivery. An optional ‘name‘ column labels the flow.

Value

A ‘bpmn‘ object: a list with ‘nodes‘, ‘flows‘, ‘name‘ and ‘messages‘ (‘NULL‘ when there are none).

Examples

```
m <- bpmn(
  nodes = data.frame(
    id = c("a", "b", "c"),
    name = c("Start", "Do work", "Done"),
    type = c("start", "task", "end")
  ),
  flows = data.frame(from = c("a", "b"), to = c("b", "c"))
)
m
```

`bpmn_to_table` *Convert a tidy BPMN model to a wide process-table scaffold*

Description

The inverse of [as_bpmn()], so a process drawn in any BPMN tool becomes a table blockr.process can run: draw in Camunda Modeler, [read_bpmn()] the file, ‘bpmn_to_table()‘ the model, drop the table into a process block.

Usage

```
bpmn_to_table(x)
```

Arguments

`x`	A [bpmn()] model.

Details

What the inversion does: - Events and gateways disappear into the ‘depends_on‘ grammar: a task behind a converging gateway depends on everything flowing into that gateway (‘join‘ records the kind – exclusive/inclusive merge becomes ‘”any”‘, a complex gateway named ‘”k of n”‘ becomes ‘”n=k”‘); a task behind a diverging gateway depends on the gateway’s input, qualified by the branch label (‘”validate:yes”‘). Chains of gateways resolve through. - Lanes become ‘role‘; a lane-less script/service/send/receive task becomes ‘role = ”system”‘. - Multi-instance activities are grouped back into containers: each connected run of multi-instance nodes gets a group row (‘collection = ”item”‘, a placeholder), its members point at it via ‘parent‘, and edges crossing the group boundary are lifted onto the group row – the inverse of the lowering [as_bpmn()] performs.

The result is a **scaffold**, not a finished definition: fill in ‘collection‘ names, check the quorums, and add ‘script‘ values where the XML did not carry them (files written by [write_bpmn()] do).

Value

A tibble with columns ‘task‘, ‘name‘, ‘role‘, ‘depends_on‘, ‘script‘, ‘collection‘, ‘parent‘, ‘join‘, ‘sequential‘.

See Also

[read_bpmn()], [as_bpmn()]

`bpmn_widget` *Render a tidy BPMN model as an interactive diagram*

Description

An htmlwidget: auto-layout (bpmn-auto-layout) and rendering (bpmn-visualization, Apache-2.0) both run in the browser, so no Node.js is needed for display. Pan and zoom are enabled.

Usage

```
bpmn_widget(
  x,
  status = NULL,
  overlays = NULL,
  width = "100%",
  height = NULL,
  elementId = NULL
)
```

Arguments

`x`	A [bpmn()] model.
`status`	Optional named character vector/list: node id -> status. ‘open‘, ‘doing‘, ‘blocked‘, ‘skipped‘ get dedicated colours; any other value (‘done‘, ‘true‘, ‘false‘, ...) is painted as finished.
`overlays`	Optional named character vector/list: node id -> small label rendered under the node (e.g. ‘”ana · 08-15”‘).
`width`, `height`	Widget size (CSS units).
`elementId`	Optional DOM id.

Value

An htmlwidget.

`bpmn_widget-shiny` *Shiny bindings for bpmn_widget*

Description

Shiny bindings for bpmn_widget

Usage

```
bpmnWidgetOutput(outputId, width = "100%", height = "420px")

renderBpmnWidget(expr, env = parent.frame(), quoted = FALSE)
```

Arguments

`outputId`	Output id.
`width`, `height`	Widget size (CSS units).
`expr`	Expression returning a [bpmn_widget()].
`env`, `quoted`	Standard shiny render arguments.

---

bpmn_xml *Generate BPMN 2.0 XML from a tidy BPMN model*

Description

Produces the semantic half of a standard BPMN 2.0 interchange document (no diagram coordinates). Use [layout_bpmn()] or [write_bpmn()] to add auto-computed layout, or [bpmn_widget()] to render directly (layout then happens in the browser).

Usage

```
bpmn_xml(x, include_lanes = TRUE)
```

Arguments

`x`	A [bpmn()] model.
`include_lanes`	Emit a ‘laneSet‘ from the ‘lane‘ column? Default ‘TRUE‘. Nodes without a lane (synthesized gateways, start/end events) inherit the lane of a neighbour, majority of predecessors first. The bundled auto-layout library (bpmn-auto-layout 1.3.0) drops all sequence-flow edges when a ‘laneSet‘ is present, so the layout paths ([layout_bpmn()] and the widget) strip the lanes before laying out and band the result back into lanes afterwards.

Value

An ‘xml2::xml_document‘.

---

code_version *Which version of the code ran*

Description

Recorded on every attempt as a ‘code_version‘ event, so the log answers ”what exactly ran here” a year later. The git commit of the jobs directory when it is a working tree, otherwise a content hash of the scripts – both are 8 hex characters, and neither claims more than ”these bytes, that day”.

Usage

```
code_version(jobs = ".")
```

Arguments

jobs	Directory holding the scripts.

Value

An 8-character string, or ‘""‘ when the directory is unreadable.

---

example_process *Example: a quarterly data collection as a wide process table*

Description

The one-table convention mid-instance: skeleton columns (‘task‘, ‘name‘, ‘role‘, ‘depends_on‘, ‘script‘) plus instance columns (‘assignee‘, ‘due‘, ‘status‘). The QA check has failed (‘status = "false"‘), so the reconciliation branch is active and approval waits.

Usage

```
example_process()
```

Value

A data frame.

---

expand_instance *Expand a definition into the instance-shaped table*

Description

One row per (element, task) for multi-instance tasks, one row per task with ‘element = NA‘ for the rest. This is what the instance store's events are folded onto and what the task lists show; the definition itself never grows.

Usage

```
expand_instance(process, elements = character(), instance = "instance")
```

Arguments

process	A wide process table (the definition).
elements	Character vector of element ids (e.g. units), or a data frame whose first column holds them plus any grouping columns (e.g. ‘region‘) to carry along.
instance	Instance id.

Value

A data frame with the definition columns plus ‘instance‘, ‘element‘ and the instance columns ‘assignee‘, ‘due‘, ‘status‘. The sub-process containers are gone – they are structure, not work – and each row carries its resolved dimension in ‘per‘, so the expanded table answers ”what does this repeat over” without the definition beside it.

---

inbox_dir *The inbox directory of a store*

Description

The inbox directory of a store

Usage

```
inbox_dir(store = ".runs")
```

Arguments

store	Store directory.

Value

The path (not created).

---

ingest_inbox *Ingest the inbox: turn waiting messages into events*

Description

Reads every ‘*.json‘ at the top level of the inbox in name order, applies the ones that make sense, and moves each file out of the way – ‘processed/‘ when it was applied (or was a duplicate of one that was), ‘failed/‘ with a ‘.error‘ file beside it when it can never apply. Messages for an instance that has not been started yet stay where they are: that is a race, not a mistake, and the next tick will get them.

Usage

```
ingest_inbox(store = ".runs", quiet = FALSE)
```

Arguments

store	Store directory.
quiet	Suppress messages.

Details

[run_worker()] calls this on every tick. Call it yourself from a cron job if your process has no scripts to run but does have an outside world.

Value

The number of messages applied, invisibly.

Examples

```
store <- tempfile()
process <- data.frame(
  task = "delivery", name = "Delivery", role = "system", depends_on = ""
)
start_instance(process, instance = "2026Q1", store = store)
write_inbox_message(store, "delivery", instance = "2026Q1",
                    actor = "upload-platform", id = "d1")
ingest_inbox(store, quiet = TRUE)
instance_log(store, "2026Q1")[, c("task", "field", "value", "actor")]
```

`instance_collection` *The element list an instance was opened with*

Description

The element list an instance was opened with

Usage

```
instance_collection(store = ".runs", instance = "instance")
```

Arguments

`store`	Store directory.
`instance`	Instance id to filter by, or ‘NULL‘ for all instances.

Value

A data frame (first column: element ids), or ‘NULL‘.

instance_definition *The definition an instance was opened with*

Description

The definition an instance was opened with

Usage

```
instance_definition(store = ".runs", instance = "instance")
```

Arguments

store	Store directory.
instance	Instance id to filter by, or ‘NULL‘ for all instances.

Value

The wide process table as stamped by [start_instance()], or ‘NULL‘ if the instance has not been opened.

instance_event *Append one event to an instance store*

Description

The only writer. Both the worker and the people write here, which is why ‘actor‘ is recorded separately from ‘assignee‘: ‘assignee‘ is whose task it is, ‘actor‘ is assignee moved it.

Usage

```
instance_event(
  task,
  field,
  value,
  actor = "worker",
  store = ".runs",
  instance = "instance",
  ts = format(Sys.time(), "%Y-%m-%dT%H:%M:%S"),
  element = NULL,
  ref = NULL
)
```

Arguments

`task`	Task id.
`field`	Field the event sets: ‘assignee‘, ‘due‘, ‘status‘, or any bookkeeping field the worker records (‘log‘, ‘took‘, ‘attempt‘).
`value`	New value.
`actor`	Assignee or what wrote this event.
`store`	Store directory.
`instance`	Instance id.
`ts`	Timestamp.
`element`	Element id for multi-instance tasks (the per-element rows, see [expand_instance()]); ‘NULL‘ for single tasks. Written to the log only when set, so single-element stores stay byte-identical.
`ref`	Where this event came from, when it came from somewhere with a name: the inbox message file, a ticket id, a request id. Written only when set. ‘actor‘ says which system spoke, ‘ref‘ says which of its messages this was – together they make an event traceable back out of the process into the system that caused it.

Value

‘value‘, invisibly.

`instance_events` *Read an instance store's events*

Description

Read an instance store's events

Usage

```
instance_events(store = ".runs", instance = NULL)
```

Arguments

`store`	Store directory.
`instance`	Instance id to filter by, or ‘NULL‘ for all instances.

Value

A data frame with columns ‘ts‘, ‘instance‘, ‘task‘, ‘field‘, ‘value‘, ‘actor‘, ‘element‘ (‘NA‘ for single-task events and old logs) and ‘ref‘ (‘NA‘ unless the event came in from a named message).

instance_latest *The newest instance in a store*

Description

The instance id of the most recently opened instance ([start_instance()] order), or ‘NULL‘ for a store without instances. The instance-reading blocks accept ‘instance = ”latest”‘ and resolve it through this on every poll – so opening a new instance moves the whole board to it, and the old instances stay in the log, addressable by their id.

Usage

```
instance_latest(store = ".runs")
```

Arguments

store	Store directory.

instance_log *An instance's events, newest first*

Description

The audit view. ‘stamp‘ exists only so a polling block's expression changes when the log grows; it is otherwise ignored.

Usage

```
instance_log(store = ".runs", instance = "instance", stamp = NULL)
```

Arguments

store	Store directory.
instance	Instance id to filter by, or ‘NULL‘ for all instances.
stamp	Poll counter (ignored).

Value

A data frame.

`instance_status` *Status of every row of an expanded instance*

Description

The multi-instance counterpart of [process_status()]. Stored status wins when it is anything but ‘open‘; for ‘open‘ rows the gates decide, resolved per the three cases above. How many gates have to be met is the row’s ‘join‘ quorum, ‘all‘ by default. Loop-back dependencies never block, same rule as the single-element case.

Usage

```
instance_status(process, instance_table)
```

Arguments

`process`	The definition (for gates, loops, ‘join‘, ‘complete_when‘ and the collection every task repeats over).
`instance_table`	An expanded instance table (see [expand_instance()]), carrying the current ‘status‘ per row.

Value

Character vector, one entry per row of ‘instance_table‘: ‘open‘ (= ready), ‘doing‘, ‘blocked‘, ‘skipped‘, or a terminal value.

`instance_table` *An instance’s expanded table, rebuilt from the store alone*

Description

[expand_instance()] over the stamped definition and elements, with the events folded on ([apply_events_instance()]). The ‘status‘ column holds the STORED status; blocked/skipped are display states, see [instance_view()].

Usage

```
instance_table(store = ".runs", instance = "instance", stamp = NULL)
```

Arguments

`store`	Store directory.
`instance`	Instance id to filter by, or ‘NULL‘ for all instances.
`stamp`	Poll counter (ignored).

Value

The expanded instance table, or an empty data frame if the instance has not been opened.

`instance_view` *An instance as people should see it*

Description

[instance_table()] with the display status computed per row: stored terminal values stay, ‘open‘ rows resolve to ‘open‘/‘blocked‘/‘skipped‘ through the dependency gates ([instance_status()]). This is what the tasks block emits and what downstream tables and the BPMN overlay read.

Usage

```
instance_view(store = ".runs", instance = "instance", stamp = NULL)
```

Arguments

`store`	Store directory.
`instance`	Instance id to filter by, or ‘NULL‘ for all instances.
`stamp`	Poll counter (ignored).

Value

The expanded instance table with ‘status‘ as display status.

`layout_bpmn` *Add auto-computed diagram layout to a BPMN model*

Description

Runs the bundled ’bpmn-auto-layout’ library (via Node.js) to add BPMN diagram interchange (shape coordinates, edge waypoints) to the semantic XML, so the exported file opens with a proper diagram in any BPMN tool.

Usage

```
layout_bpmn(x)
```

Arguments

`x`	A [bpmn()] model or a BPMN XML string.

Value

BPMN XML (character) including ‘BPMNDiagram‘ layout.

`new_assign_block` *Instance block (JS-driven)*

Description

The skeleton brought to life: receives the wide process table, lets people assign ‘assignee‘, ‘due‘ and advance ‘status‘ per task, and emits the table with the instance columns applied. Statuses ‘blocked‘ and ‘skipped‘ are computed from the dependency gates, never stored. Tasks whose outcome other tasks consume (checks) finish with an outcome label (‘true‘/‘false‘ by convention) instead of ‘done‘.

Usage

```
new_assign_block(assignments = list(), ...)
```

Arguments

`assignments`	Named list: task id -> ‘list(assignee =, due =, status =)‘.
`...`	Forwarded to [blockr.core::new_transform_block()].

Details

The control UI is a JavaScript task list (‘inst/js/assign-block.js‘); R receives the assignments as JSON and turns them into an [apply_assignments()] expression.

‘assignee‘ is a free-text field unless the incoming column is a **factor**, in which case its levels are the roster and the field becomes a picker. The allowed values are a property of the data, not of this block: declare them upstream with a stock mutate block, ‘assignee = factor(NA_character_, levels = c(”ana”, ”ben”))‘.

`new_bpmn_block` *BPMN block*

Description

Renders the incoming wide process table as a BPMN diagram (via the BPMN half: auto-layout + bpmn-visualization, no watermark) with the instance painted on: finished tasks green, ‘doing‘ blue, computed ‘blocked‘ amber and ‘skipped‘ gray, and a ‘assignee - due‘ overlay under assigned tasks. The data passes through unchanged, so downstream blocks (tables, dplyr views) keep working on the same table.

Usage

```
new_bpmn_block(title = "Process", ...)
```

Arguments

`title`	Process name shown in the exported BPMN.
`...`	Forwarded to [blockr.core::new_transform_block()].

`new_event_log_block` *Instance log block*

Description

An instance's events, newest first: the audit history. Nothing here is ever updated or deleted; the instance columns upstream are a fold over exactly these rows.

Usage

```
new_event_log_block(store = ".runs", instance = "instance", poll = 2, ...)
```

Arguments

`store`	Instance store directory (see [instance_event()]).
`instance`	Instance id, or '"latest"' to follow the newest instance in the store.
`poll`	Poll interval in seconds.
`...`	Forwarded to [blockr.core::new_data_block()].

`new_process_block` *Process block (JS-driven editor)*

Description

Design a process as a list of tasks: name, role, dependencies (with optional outcome qualifiers – pick "QA check · only true" to hang a task off one branch of a check) and, for 'system' tasks, the script to run. The block emits the wide process table; the diagram and the instance views downstream redraw as you edit.

Usage

```
new_process_block(tasks = example_tasks(), ...)
```

Arguments

`tasks`	List of tasks, each 'list(task =, name =, role =, dep =, script =, parent =, collection =)' where 'dep' is the comma-separated 'depends_on' string (outcome qualifiers as 'task:label'), 'collection' marks the row as a multi-instance sub-process and names what it repeats over, and 'parent' puts a task inside one. Defaults to the quarterly production example.
`...`	Forwarded to [blockr.core::new_data_block()].

Details

The control UI is the outline rail (‘blockr.outline::outline_rail_dep()‘) driven by a process adapter (‘inst/js/process-block.js‘): tasks are rows in flow order, dependencies are the rail drawn beside them, and the gestures are the board editor’s – drag a task’s dot onto another row to make it depend on it, release on empty canvas to append a task, hover a rail edge for the .

The part of a process that repeats is a group: select the rows, press ”Group them”, and they sit inside a multi-instance sub-process whose header names the collection it runs once per. That frame is not a UI invention – it is how BPMN models a fan-out – and it is the only place the collection is named, so two tasks can no longer disagree about what the process repeats over.

`new_runstate_block` *Instance state block*

Description

Overlays an instance store onto the incoming process definition: the instance columns (‘assignee‘, ‘due‘, ‘status‘) come from the append-only log, so the board shows what the worker and the people have actually done, live.

Usage

```
new_runstate_block(store = ".runs", instance = "instance", poll = 2, ...)
```

Arguments

`store`	Instance store directory (see [instance_event()]).
`instance`	Instance id, or ‘”latest”‘ to follow the newest instance in the store.
`poll`	Poll interval in seconds.
`...`	Forwarded to [blockr.core::new_transform_block()].

Details

Polling is on the log’s size and mtime, so an idle board redraws nothing. The poll counter rides along in the block’s expression as ‘stamp‘, which is what makes the board re-evaluate when the log grows. A database-backed store would push instead.

`new_start_instance_block` *Open-instance block*

Description

Takes the process definition as its input and opens an instance from it: pick the year (the instance id), pick a collection, click "Start instance" – one [start_instance()]. Once the instance exists the card shows its header (id, [process_version()], created, size) and notes when the incoming definition no longer matches the stamped one.

Usage

```
new_start_instance_block(
  store = ".runs",
  instance = "instance",
  source = "",
  sources = list(),
  poll = 2,
  class = "start_instance_block",
  ctor = sys.parent(),
  ctor_pkg = NULL,
  ...
)
```

Arguments

`store`	Instance store directory (see [instance_event()]).
`instance`	Instance id.
`source`	Name of the selected element list.
`sources`	Named list of element lists: each a data frame (first column: element ids, e.g. units; further columns become the instance table's facets, an 'assignee' column seeds assignments) or a zero-argument function returning one (a 'SELECT' on the platform's database). Not serialized: a deployment wraps this constructor with its lists baked in and registers the wrapper.
`poll`	Poll interval in seconds.
`class`	Block S3 class. The wrapping constructor passes its own subclass (e.g. 'c("start_block", "start_instance_block")') so the registry keys on the wrapper.
`ctor`, `ctor_pkg`	Forwarded to [blockr.core::new_transform_block()]; default: the CALLING constructor becomes the ctor of record, so its formals ('store', 'instance', 'source', 'poll') define the block's state.
`...`	Forwarded to [blockr.core::new_transform_block()].

Details

The block emits [instance_view()], the expanded instance table.

Not registered directly

This is a factory in the spirit of the JS-block factories: ‘sources‘ holds data (or query functions) that cannot serialize, so a deployment MUST wrap it in its own constructor – with the lists baked in – and register that wrapper. See the demo's ‘dcdemo::new_start_block()‘.

`new_tasks_block` *Tasks block*

Description

An instance's task list: one section per task, one row per element for the "per-element" tasks, with per-row assignment and a status chip that writes [instance_event()]s. The toolbar filters (status, person, any grouping column the element list carried, e.g. ‘region‘) and acts on the selection: bulk-assign.

Usage

```
new_tasks_block(
  store = ".runs",
  instance = "instance",
  poll = 2,
  roster = character(),
  ...
)
```

Arguments

`store`	Instance store directory (see [instance_event()]).
`instance`	Instance id, or ‘"latest"‘ to follow the newest instance in the store ([instance_latest()]): opening a new instance moves the list to it.
`poll`	Poll interval in seconds.
`roster`	Characters: the people assignments can pick from (the deployment's register). The definition's roles are NOT offered – roles are lanes, not persons; without a roster the block falls back to them so a bare demo stays usable.
`...`	Forwarded to [blockr.core::new_data_block()].

Details

The block carries no action-specific gesture beyond that. ”Send this back” is a decision, and decisions belong in the definition as outcomes: a dependent qualifies its predecessor (‘fix depends_on review:rejected‘), the chip then offers ‘rejected‘ as a terminal state, and the branch is visible in the diagram instead of hidden in a button. Reopening a task that was already finished is one event from outside – [process_act()], an inbox message, a mirror – because the board is where open work is acted on, not where history is rewritten.

The block emits [instance_view()], so downstream tables and the BPMN diagram redraw on every event. Polling is on the log’s size and mtime; an idle board redraws nothing.

---

process_act *A person acts on a task*

Description

The other writer of the instance store. This is exactly what the tasks block’s status chip calls, and what a command line or an inbox message ends up calling: the worker cannot tell the difference.

Usage

```
process_act(
  task,
  status = "done",
  assignee = NULL,
  due = NULL,
  actor = NULL,
  store = ".runs",
  instance = "instance",
  process = NULL,
  element = NULL
)
```

Arguments

task	Task id.
status	New status (‘doing‘, ‘done‘, or a check’s outcome label).
assignee	Person the task is assigned to.
due	Deadline.
actor	Who wrote the event; defaults to ‘assignee‘.
store	Store directory.
instance	Instance id.
process	The definition, needed to re-arm loops.
element	Element id when the task is multi-instance; ‘NULL‘ for single tasks.

Value

'status', invisibly.

`process_groups` *Multi-instance groups of a wide process table*

Description

The part of a process that repeats is a BPMN **multi-instance sub-process**, and under the one-table convention that sub-process is a row like any other: a row with a non-empty 'collection' is the group, and the rows whose 'parent' names it are its members. Each member runs once per element of the collection, and a task outside the group that depends on it waits for the whole sub-process to complete (the parallel join, see 'complete_when' for the quorum).

Usage

```
process_groups(df)

process_scopes(df)

process_collection(df)

process_complete_when(df)

process_sequential(df)

process_body(df)
```

Arguments

`df`	A wide process table.

Details

The value of 'collection' names the **collection type** the sub-process iterates ('unit'), not the concrete list: the definition is list-agnostic and the elements are bound when an instance starts. In Camunda's vocabulary the column therefore holds 'inputElement', while 'inputCollection' is instance data.

'process_groups()' lists the groups, 'process_collection()' answers the question every consumer actually asks – what does this task repeat over – and 'process_body()' drops the containers, leaving the rows that are real work.

Groups nest: a group whose 'parent' names another group repeats inside it. 'process_scopes()' gives the full enclosing chain, 'process_collection()' the innermost collection.

Tables written before groups existed marked every repeated task with a 'per' column instead; 'process_collection()' falls back to it when there is no 'collection' column, so old definitions (and the process JSON stamped into a running instance) keep folding.

BPMN forbids a sequence flow that crosses a sub-process boundary: the container carries the incoming and outgoing flows, its members are wired only among themselves. ‘process_body()‘ is where that is turned back into a flat table, by *lowering* each container’s edges onto its members, innermost group first:

- ‘X depends_on G‘ becomes a dependency on every **leaf** member of ‘G‘ (the members nothing else in ‘G‘ waits for), which is the parallel join where the sub-process closes. - ‘G depends_on Y‘ becomes a dependency of every **root** member of ‘G‘ (the members that wait for nothing else in ‘G‘) on ‘Y‘, which is the fan-out where it opens.

A dependency that names a *member* from outside the group is left alone. It lowers to the same edge, which is why definitions written before containers were addressable keep working unchanged.

Value

‘process_groups()‘: a data frame with ‘group‘, ‘name‘, ‘collection‘, ‘parent‘ (the enclosing group, ‘""‘ at top level), ‘complete_when‘ and ‘sequential‘, one row per group.

‘process_scopes()‘: a named list over every row of ‘df‘ – the group ids enclosing that row, innermost first, ‘character(0)‘ at top level. Errors on a ‘parent‘ cycle.

‘process_collection()‘: a named character vector over every row of ‘df‘ – the collection that row repeats over, ‘""‘ for single tasks and for the group rows themselves. Nested groups resolve to the innermost.

‘process_complete_when()‘: a named character vector over every row – the quorum spec of the group that row belongs to (‘""‘ = all, and ‘""‘ for single tasks). This is what a task waiting on the group has to clear; see [process_quorum()] for the grammar.

‘process_sequential()‘: a named logical vector over every row – whether the group that row belongs to runs its elements one at a time (BPMN ‘isSequential‘). ‘FALSE‘ for single tasks.

‘process_body()‘: ‘df‘ without its group rows, with every container edge lowered onto the members (see details).

`process_interpreters` *How a script file is executed, by extension*

Description

The worker runs files, not commands: the ‘script‘ column of a process table names a file inside the jobs directory, and its extension picks the interpreter. Pass your own list to [run_worker()] to add one (‘list(jl = list(command = "julia", args = character()))‘); the names are lowercase extensions.

Usage

```
process_interpreters()
```

Details

'Rscript –vanilla' on purpose: a script that needs a package environment should say so itself ('renv::load()' on line one), so that what ran is visible in the script rather than in the worker's shell.

Value

A named list of 'list(command, args)'.

`process_status` *Computed display status for every task of a wide process table*

Description

Stored status wins when it is anything but 'open' ('doing', 'done', or an outcome label – any non-open value means the task has life). For 'open' tasks the gates decide: a gate whose dependency finished with the outcome it asked for is met, one that finished with another outcome never will be, and an unfinished one may still go either way. How many have to be met is the row's 'join' quorum, 'all' by default ([process_quorum()]). Loop-back dependencies never block.

Usage

```
process_status(df)
```

Arguments

`df`	A wide process table.

Value

Named character vector: task -> display status.

`process_tasks` *Per-task metadata for a wide process table*

Description

The machine-readable reading of the table that run UIs and status computation share: for every task its gates (non-loop dependencies with optional outcome label), loop-back dependencies, and the outcome labels other tasks consume from it (non-empty marks the task as a check).

Usage

```
process_tasks(df)
```

Arguments

df	A wide process table (see [as_bpmn()] for the conventions).

Value

A list (one element per task) of lists with ‘task‘, ‘name‘, ‘role‘, ‘script‘, ‘per‘ (the collection it repeats over, ‘""‘ for single tasks), ‘join‘ (the fan-in quorum spec), ‘assignee‘, ‘due‘, ‘status‘, ‘gates‘ (list of ‘list(on, label)‘), ‘loops‘ (character), ‘outcomes‘ (character). Multi-instance group rows are containers, not tasks, and are left out.

---

process_version *Short content hash of a process definition*

Description

Polynomial rolling hash over the definition's cells; enough to tell two versions of a process apart in an audit line, not a cryptographic commitment. Deterministic across sessions.

Usage

```
process_version(process)
```

Arguments

process	A wide process table.

Value

An 8-character hex string.

---

read_bpmn *Read a BPMN 2.0 XML file into a tidy BPMN model*

Description

The inverse of [bpmn_xml()]: parses the semantic half of a BPMN 2.0 interchange document – from this package, Camunda Modeler, bpmn.io or any other standards-conform tool – into a [bpmn()] model. Diagram coordinates are ignored (layout is always recomputed), namespace prefixes don't matter (‘bpmn:‘, ‘bpmn2:‘, none).

Usage

```
read_bpmn(x)
```

Arguments

x	A path to a ‘.bpmn‘ file, a string of BPMN XML, or an ‘xml2::xml_document‘.

Details

What is read: the flow nodes [bpmn()] models (events, tasks, gateways), sequence flows with their labels, lanes, multi-instance markers (‘isSequential‘ included), a ‘<script>‘ child on script tasks (the worker wiring [bpmn_xml()] writes), and message flows to or from collapsed pools. A file using elements outside that vocabulary (sub-processes, boundary or intermediate events, call activities) is refused with a list of the offending elements rather than silently flattened.

Value

A [bpmn()] model. Pass it on to [bpmn_to_table()] to get a wide process-table scaffold blockr.process can run.

See Also

[bpmn_to_table()], [bpmn_xml()], [write_bpmn()]

`run_worker` *Run a process until it needs a person (or finishes)*

Description

The worker is a plain R process: start it from cron, a systemd unit, a Connect scheduled job or a second terminal. It holds no state of its own – everything it knows it read from the store, everything it did it wrote there – so killing it and starting it again loses nothing but the time in between.

Usage

```
run_worker(
  process,
  store = ".runs",
  instance = "instance",
  jobs = ".",
  wait = TRUE,
  tick = 2,
  timeout = 900,
  lock = TRUE,
  inbox = TRUE,
  interpreters = process_interpreters(),
  packages = character(),
  quiet = FALSE
)
```

Arguments

process	A wide process table (the definition).
store	Instance store directory.
instance	Instance id, or '"latest"' for the newest instance in the store.
jobs	Directory holding the scripts named in the 'script' column. Scripts are resolved inside it and nowhere else.
wait	Poll for human events instead of returning at the first task a person owns.
tick	Poll interval in seconds.
timeout	Give up waiting after this many seconds.
lock	Take the worker lock, so a second worker on the same store refuses to start.
inbox	Ingest the file inbox on every tick.
interpreters	Extension -> interpreter map, see [process_interpreters()].
packages	Packages whose functions a 'script' cell may name ('"mypkg::forecast"'). The allowlist for the function form, and the counterpart of 'jobs' for the file form: empty by default, so a deployment has to say which code it is willing to run.
quiet	Suppress progress messages.

Details

On every tick it ingests the inbox ([ingest_inbox()]), folds the log, and runs the ready tasks whose role is 'system' and which name a script. A ready task belonging to a person stops it (or, with 'wait = TRUE', it waits for that person's event).

Value

The table with the instance applied, invisibly.

Examples

```
store <- tempfile()
jobs <- tempfile()
dir.create(jobs)
writeLines("cat('hello\n')", file.path(jobs, "hello.R"))

process <- data.frame(
  task = "greet", name = "Greet", role = "system",
  depends_on = "", script = "hello.R"
)
run_worker(process, store = store, jobs = jobs, wait = FALSE, quiet = TRUE)
instance_log(store, "instance")[, c("task", "field", "value")]
```

`start_instance` *Open an instance: stamp definition and elements into the store*

Description

Writes the '@instance' events that make an instance self-contained: the definition as opened (field 'process'), the element list (field 'elements'), the definition's [process_version()] (field 'version') and a 'created' mark. Everything after this moment is ordinary events on the instance's rows.

Usage

```
start_instance(
  process,
  elements = character(),
  instance = "instance",
  store = ".runs",
  actor = "board"
)
```

Arguments

`process`	A wide process table (the definition).
`elements`	Character vector of element ids, or a data frame whose first column holds them plus grouping columns (e.g. 'region') carried into the instance table. An 'assignee' column seeds the assignment: every human multi-instance task of that element gets a 'assignee' event (actor 'register') – the "assignment comes from the register" answer. A list without the column leaves everything in the pool.
`instance`	Instance id (a period: "2026Q1", "2026").
`store`	Store directory.
`actor`	Assignee opened the instance.

Value

The instance id, invisibly.

`worker_alive` *Is a worker running on this store?*

Description

Reads the lock a running [run_worker()] holds and touches on every tick. A board can show it ("no worker is running, scripts will not start"), and a script that would rather ingest the inbox itself can ask first.

Usage

```
worker_alive(store = ".runs", stale = 120)
```

Arguments

store	Store directory.
stale	Seconds without a heartbeat after which the answer is ‘FALSE‘.

Value

‘TRUE‘ or ‘FALSE‘.

---

write_bpmn *Write a tidy BPMN model to a .bpmn file*

Description

Write a tidy BPMN model to a .bpmn file

Usage

```
write_bpmn(x, path, layout = TRUE)
```

Arguments

x	A [bpmn()] model.
path	Output file path (conventionally ‘.bpmn‘).
layout	Add auto-computed diagram layout (needs Node.js)? If ‘FALSE‘, only the semantic XML is written.

Value

‘path‘, invisibly.

`write_inbox_message` *Write one inbox message*

Description

The R-side sender, useful from a scheduled script, a test, or a demo block standing in for a real platform. Any other language does the same thing by writing the same JSON: write to a temporary name in the same directory, then rename, so the ingester never sees a half-written file.

Usage

```
write_inbox_message(
  store = ".runs",
  task,
  value = "done",
  field = "status",
  element = NULL,
  instance = "latest",
  actor = "external",
  id = NULL
)
```

Arguments

`store`	Store directory.
`task`	Task the message is about.
`value`	New value (default ‘"done"‘).
`field`	Field it sets (default ‘"status"‘).
`element`	Element id for a multi-instance task; ‘NULL‘ for single tasks.
`instance`	Instance id, or ‘"latest"‘.
`actor`	Who is speaking – the name of the sending system. It ends up in the audit trail, so use the system's name, not a person's.
`id`	Message id. Becomes the file name and is the idempotency key; defaults to a timestamped one. Give it a stable, meaningful id (‘delivery-2026Q1-northside‘) and a resend is free.

Value

The path written, invisibly.

Examples

```
store <- tempfile()
write_inbox_message(store, "delivery", element = "northside",
                    actor = "upload-platform", id = "delivery-northside")
```

Index

apply_events, 3
as_bpmn, 3

bpmn, 4
bpmn_to_table, 6
bpmn_widget, 7
bpmn_widget-shiny, 7
bpmn_xml, 8
bpmnWidgetOutput *(bpmn_widget-shiny)*, 7

code_version, 8

example_process, 9
expand_instance, 9

inbox_dir, 10
ingest_inbox, 10
instance_collection, 11
instance_definition, 12
instance_event, 12
instance_events, 13
instance_latest, 14
instance_log, 14
instance_status, 15
instance_table, 15
instance_view, 16

layout_bpmn, 16

new_assign_block, 17
new_bpmn_block, 17
new_event_log_block, 18
new_process_block, 18
new_runstate_block, 19
new_start_instance_block, 20
new_tasks_block, 21

process_act, 22
process_body *(process_groups)*, 23
process_collection *(process_groups)*, 23
process_complete_when *(process_groups)*, 23
process_groups, 23
process_interpreters, 24
process_scopes *(process_groups)*, 23
process_sequential *(process_groups)*, 23
process_status, 25
process_tasks, 25
process_version, 26

read_bpmn, 26
renderBpmnWidget *(bpmn_widget-shiny)*, 7
run_worker, 27

start_instance, 29

worker_alive, 29
write_bpmn, 30
write_inbox_message, 31